Designing XML API for Modern C++

Boris Kolpackov

Code Synthesis

v1.9, May 2014

XML and C++

XML and C++

- Not in standard C++
- Not in Boost
- Existing libraries “not great”

Talk Outline

- Why XML?
- Terminology
- Related tech
- Real XML parsers
- Common XML APIs
- Existing tools and libraries

Talk Outline

- XML usage patterns in C++
- New API and examples
- Your feedback
- Implementation details

Who is this Guy?

10 years of XML in C++

XML Quiz

Q: Who publishes the worst XML vocabularies?

XML Quiz

Q: Who publishes the worst XML vocabularies?

A: W3C

Why XML?

1. XML is an unnecessary evil
2. XML is sometimes the right answer
3. XML is the best thing since sliced bread

Why Use XML?

Interchange, not just Data Storage

- Accessibility
- Tooling
- Education

Why Use XML?

Interchange, not just Data Storage

- Accessibility
- Tooling
- Education
- Human Read/Write-able

What's the Goal?

Say No to XML Frameworks

What's the Goal?

Say No to XML Frameworks

Rather, Get Away from XML, Fast

What’s the Goal?

- Open source
- Cross platform
- Compact
- External dependecy-free

What's the Goal?

- Open source
- Cross platform
- Compact
- External dependecy-free

Base for `boost::xml` or `std::xml`?

XML Vocabulary

Specialization of XML

Object Model

C++ classes that represent XML data

Data-Centric Vocabularies

XML is just a medium for storing the data

Document-Centric Vocabularies

XML structure as important as data

Content Model

```
<empty name="a" id="1"/>

<simple name="b" id="2">text<simple/>

<complex name="c" id="3">
  <nested>...</nested>
  <nested>...</nested>
<complex/>

<mixed name="d" id="4">
  te<nested>...</nested>
  x
  <nested>...</nested>t
<mixed/>
```

Content Model

```
<empty name="a" id="1"/>

<simple name="b" id="2">text<simple/>

<complex name="c" id="3">
  <nested>...</nested>
  <nested>...</nested>
<complex/>

<mixed name="d" id="4">
  te<nested>...</nested>
  x
  <nested>...</nested>t
<mixed/>
```

Content Model

```
<empty name="a" id="1"/>

<simple name="b" id="2">text<simple/>

<complex name="c" id="3">
  <nested>...</nested>
  <nested>...</nested>
<complex/>

<mixed name="d" id="4">
  te<nested>...</nested>
  x
  <nested>...</nested>t
<mixed/>
```

Content Model

```
<empty name="a" id="1"/>

<simple name="b" id="2">text<simple/>

<complex name="c" id="3">
  <nested>...</nested>
  <nested>...</nested>
<complex/>

<mixed name="d" id="4">
  te<nested>...</nested>
  x
  <nested>...</nested>t
<mixed/>
```

Content Model

```
<empty name="a" id="1"/>

<simple name="b" id="2">text<simple/>

<complex name="c" id="3">
  <nested>...</nested>
  <nested>...</nested>
<complex/>

<mixed name="d" id="4">
  te<nested>...</nested>
  x
  <nested>...</nested>t
<mixed/>
```

XML-Related Technologies

Useful:	Core XML, Namespaces, XML Schema, XPath
Useless:	XLink, XInclude, XPointer
”Life is XML”:	XQuery, XSLT, XProc, XForms

XML Schema

```
<person id="1">
  <first>Joe</first>
  <last>Dirt</last>
  <age>23</age>
</person>

<xs:complexType name="person_type">
  <xs:sequence>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
    <xs:element name="age" type="xs:unsignedShort"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:unsignedLong"/>
</xs:complexType>

<xs:element name="person" type="person_type"/>
```

XML-Related Technologies

Useful:	Core XML, Namespaces, XML Schema, XPath
Useless:	XLink, XInclude, XPointer
”Life is XML”:	XQuery, XSLT, XProc, XForms

Real XML Parser

- At least UTF-8 and UTF-16 encodings
- CDATA sections (`<![CDATA[<hello/>]]>`)
- Character references (`&#x20;`)
- Entity references, including predefined (`&amp;`) and user-defined in the internal DTD subset
- Parse and check for well-formedness the internal DTD subset
- Normalize and supply default attribute values according to the internal DTD subset

Real XML Parser

```
<!DOCTYPE person [
<!ENTITY joe ”Joe”>
<!ATTLIST first lang NMTOKEN ”en”>
<!ATTLIST last lang NMTOKEN ”en”>
]>
<person>
  <first lang=”  fr  ”>%joe;</first>
  <last>Dirt&#x20;&amp;&#x20;<![CDATA[Snow]]></last>
</person>
```

Real XML Parser

```
<!DOCTYPE person [
<!ENTITY joe ”Joe”>
<!ATTLIST first lang NMTOKEN ”en”>
<!ATTLIST last lang NMTOKEN ”en”>
]>
<person>
  <first lang=”  fr  ”>%joe;</first>
  <last>Dirt&#x20;&amp;&#x20;<![CDATA[Snow]]></last>
</person>


<person>
  <first lang=”fr”>Joe</first>
  <last lang=”en”>Dirt & Snow</last>
</person>
```

In-Memory XML Processing

```
                    DOMDocument
<person id="1">     DOMElement("person", {"id","1"})
  <name>              DOMElement("name")
    Joe Dirt            DOMText("Joe Dirt")
  </name>
  <age>               DOMElement("age")
    23                  DOMText("23")
  </age>
</person>
```

In-Memory XML Processing

Advantages:

- All the data accessible all the time

In-Memory XML Processing

Advantages:

- All the data accessible all the time

Disadvantages:

- Document may not fit into memory
- Performance cost of in-memory representation

Streaming XML Processing

```
<person id="1">
  <name>
    Joe Dirt
  </name>
  <age>
    23
  </age>
</person>
```

```
start_element("person", {"id", "1"})
  start_element("name")
    characters("Joe Dirt")
  end_element
  start_element("age")
    characters("23")
  end_element
end_element
```

Streaming XML Processing

Advantages:

- Faster than in-memory
- Requires less memory

Streaming XML Processing

Advantages:

- Faster than in-memory
- Requires less memory

Disadvantages:

- Only see a small fragment of XML at a time

Streaming XML Parsing

- Push model: the parser calls you (callbacks; SAX)
- Pull model: you call the parser (iteration)

Hybrid XML Processing

```
<people>     start_element(”people”))
  <person>     DOMElement(”person”)
    ...          ...
  </person>
  <person>     DOMElement(”person”)
    ...          ...
  </person>
  ...          ...
</people     end_element
```

XML Data Binding

```
<person                          class person {
    id="1">                        unsigned long id ();
  <name>Joe Dirt</name>            std::string name ();
  <age>23</age>                    unsigned short age ();
</person>                        };
```

Existing XML Libraries and Tools

Real:

- Xerces-C++
- libxml2
- Expat
- Qt

Subset:

- RapidXML
- TinyXML
- *XML (until proven otherwise)

Xerces-C++

”Enterprise XML parser for C++”

License:	Apache
Language:	C++
APIs:	SAX, DOM, XPath (subset)
Validation:	DTD and XML Schema

libxml2

”Object-oriented programming in C”

License: MIT

Language: C

APIs: SAX, DOM, Pull parser, XPath

Validation: DTD

Expat

”Do one thing but do it right”

License:	MIT
Language:	C
APIs:	SAX
Validation:	No

Qt

”The Qt way”

License: LGPLv2 + commercial

Language: C++

APIs: Pull parser/serializer, SAX and DOM (deprecated)

Validation: No

XML Data Binding Tools

- CodeSynthesis XSD
- CodeSynthesis XSD/e
- XmlPlus

CodeSynthesis XSD

License:	GPLv2 + commercial
Input:	XML Schema
Models:	In-memory, streaming, hybrid
Base:	Xerces-C++
Validation:	yes (Xerces-C++)

CodeSynthesis XSD/e

License:	GPLv2 + commercial
Input:	XML Schema
Models:	In-memory, streaming, hybrid
Base:	Expat
Validation:	yes (generated code)

XmlPlus

License: LGPLv3

Input: XML Schema

Models: In-memory

Base: POCO, Expat, PCRE, custom DOM

Validation: yes? (claimed)

Other Ways to Handle XML

- Boost Property Tree
- Boost Serialization

Other Ways to Handle XML

- Boost Property Tree
- Boost Serialization

Little control over XML vocabulary

XML Usage Patterns

What are common XML usage patterns in C++?

- Not interested in outliers
- Using XML as interchange format

XML Usage Patterns

- Object persistence (config files, test setups, etc)
- Messaging (XML-RPC, SOAP)
- XML converters/filters
- Document-centric XML (OpenOffice XML, XHTML)

XML Usage Patterns

Common:

- Object persistence (config files, test setups, etc)
- Messaging (XML-RPC, SOAP)

Outliers:

- XML converters/filters
- Document-centric XML (OpenOffice XML, XHTML)

XML Usage Patterns

- Object persistence
- Messaging
- XML converters/filters

Common XML Vocabularies

Data or Document-centric?

Common XML Vocabularies

Data or Document-centric?

- Persistence and Messaging — normally data-centric
- Converters and Filters — normally don’t care

Common XML Vocabularies

No mixed content

Processing Model

Which processing model?

- In-memory
- Streaming

Processing Model

Which processing model?

- In-memory
- Streaming

What application does with the data?

Processing Model

What application does with the data?

- Initialize state of a C++ class based on XML data
- Compute something based on XML data
- Convert/filter XML

Processing Model

```
<object id="123">
  <name>Lion's Head</name>
  <position lat="-33.8569" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5083"/>
  ...
</object>
```

- C++ classes save themselves to XML

Processing Model

```
<object id="123">
  <name>Lion's Head</name>
  <position lat="-33.8569" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5083"/>
  ...
</object>
```

- C++ classes save themselves to XML
- ...

- ...
- Compute without object model

Processing Model

```
<object id="123">
  <name>Lion's Head</name>
  <position lat="-33.8569" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5083"/>
  ...
</object>
```

- C++ classes save themselves to XML
- ...
- Compute + object model
- ...
- Compute without object model

Processing Model

Which processing model?

- In-memory
- Streaming

Processing Model

Push or Pull parsing?

- Convenience: pull
- Raw speed: push

Processing Model

Push or Pull parsing?

- Convenience: pull
- Raw speed: push

XML wrong choice for max speed

Processing Model

Summary

Vocabulary:	data-centric
Content:	predominantly simple or complex
Processing:	streaming
Parsing API:	pull

API

```
#include <xml/parser>
#include <xml/serializer>

namespace xml
{
  class parser;
  class serializer;
}
```

Low Level Parser API

```
class parser
{
  typedef unsigned short feature_type;

  static const feature_type receive_elements;
  static const feature_type receive_characters;
  static const feature_type receive_attributes;
  static const feature_type receive_namespace_decls;

  static const feature_type receive_default =
    receive_elements |
    receive_characters |
    receive_attributes;

  parser (std::istream&,
          const std::string& input_name,
          feature_type = receive_default);
  ...
};
```

Low Level Parser API

```
class parser
{
  typedef unsigned short feature_type;

  static const feature_type receive_elements;
  static const feature_type receive_characters;
  static const feature_type receive_attributes;
  static const feature_type receive_namespace_decls;

  static const feature_type receive_default =
    receive_elements |
    receive_characters |
    receive_attributes;

  parser (std::istream&,
          const std::string& input_name,
          feature_type = receive_default);
  ...
};
```

Low Level Parser API

```
class parser
{
  typedef unsigned short feature_type;

  static const feature_type receive_elements;
  static const feature_type receive_characters;
  static const feature_type receive_attributes;
  static const feature_type receive_namespace_decls;

  static const feature_type receive_default =
    receive_elements |
    receive_characters |
    receive_attributes;

  parser (std::istream&,
          const std::string& input_name,
          feature_type = receive_default);
  ...
};
```

Low Level Parser API

```
class parser
{
  typedef unsigned short feature_type;

  static const feature_type receive_elements;
  static const feature_type receive_characters;
  static const feature_type receive_attributes;
  static const feature_type receive_namespace_decls;

  static const feature_type receive_default =
    receive_elements |
    receive_characters |
    receive_attributes;

  parser (std::istream&,
          const std::string& input_name,
          feature_type = receive_default);

  ...
};
```

XML Filter Example

```
int main (int argc, char* argv[])
{
  ...

  using namespace xml;

  ifstream ifs (argv[1]);
  parser p (ifs, argv[1]);

  ...
}
```

Low Level Parser API

```
class parser
{
  enum event_type
  {
    start_element,
    end_element,
    start_attribute,
    end_attribute,
    characters,
    start_namespace_decl,
    end_namespace_decl,
    eof
  };

  event_type next ();
};
```

Low Level Parser API

```
class parser
{
  enum event_type
  {
    start_element,
    end_element,
    start_attribute,
    end_attribute,
    characters,
    start_namespace_decl,
    end_namespace_decl,
    eof
  };

  event_type next ();
};
```

XML Filter Example

```
for (parser::event_type e (p.next ());
     e != parser::eof;
     e = p.next ())
{
  switch (e)
  {
  case parser::start_element:
    ...
  case parser::end_element:
    ...
  case parser::start_attribute:
    ...
  case parser::end_attribute:
    ...
  case parser::characters:
    ...
  }
}
```

XML Filter Example (C++11)

```
for (parser::event_type e: p)
{
  switch (e)
  {
    ...
  }
}
```

Low Level Parser API

```
class parser
{
  const std::string& name () const;
  const std::string& value () const;

  unsigned long long line () const;
  unsigned long long column () const;
};
```

Low Level Parser API

```
class parser
{
  const std::string& name () const;
  const std::string& value () const;

  unsigned long long line () const;
  unsigned long long column () const;
};
```

Low Level Parser API

```
class parser
{
  const std::string& name () const;
  const std::string& value () const;

  unsigned long long line () const;
  unsigned long long column () const;
};
```

Low Level Parser API

```
class parser
{
  const std::string& name () const;
  const std::string& value () const;

  unsigned long long line () const;
  unsigned long long column () const;
};
```

XML Filter Example

```
switch (e)
{
case parser::start_element:
  cerr << p.line () << ':' << p.column () << ": start "
       << p.name () << endl;
  break;
case parser::end_element:
  cerr << p.line () << ':' << p.column () << ": end "
       << p.name () << endl;
  break;
}
```

Low Level Serializer API

```
class serializer
{
  serializer (std::ostream&,
              const std::string& output_name,
              unsigned short indentation = 2);

  ...
};
```

Low Level Serializer API

```
class serializer
{
  serializer (std::ostream&,
              const std::string& output_name,
              unsigned short indentation = 2);

  ...
};
```

XML Filter Example

```
int main (int argc, char* argv[])
{
  ...

  using namespace xml;

  ifstream ifs (argv[1]);
  parser p (ifs, argv[1]);
  serializer s (cout, ”output”, 0);

  ...
}
```

Low Level Serializer API

```
class serializer
{
  void start_element (const std::string& name);
  void end_element ();

  void start_attribute (const std::string& name);
  void end_attribute ();

  void characters (const std::string& value);
};
```

XML Filter Example

```
bool skip (false);

for (parser::event_type e: p)
{
  switch (e)
  {
  case parser::start_element:
    s.start_element (p.name ());
    break;
  case parser::end_element:
    s.end_element ();
    break;
```

XML Filter Example

```
  case parser::start_attribute:
    if (p.name () == "id")
      skip = true;
    else
      s.start_attribute (p.name ());
    break;
  case parser::end_attribute:
    if (skip)
      skip = false;
    else
      s.end_attribute ();
    break;
  case parser::characters:
    if (!skip)
      s.characters (p.value ());
    break;
  }
}
```

Qualified Names

```
namespace xml
{
  class qname
  {
  public:
    qname ();
    qname (const std::string& name);
    qname (const std::string& namespace_,
           const std::string& name);

    const std::string& namespace_ () const;
    const std::string& name () const;
  };
}
```

Qualified Names

```
class parser
{
  const qname& qname () const;
};

class serializer
{
  void start_element (const qname&);
  void start_attribute (const qname&);
};
```

Namespace-Aware XML Filter Example

```
switch (e)
{
case parser::start_element:
  s.start_element (p.qname ());
  break;
case parser::start_attribute:
  if (p.qname () == "id") // Unqualified name.
    skip = true;
  else
    s.start_attribute (p.qname ());
  break;
}
```

Namespace-Aware XML Filter Example

```
parser p (ifs,
          argv[1]
          parser::receive_default |
          parser::receive_namespace_decls);

for (...)
{
  switch (e)
  {
    ...

  case parser::start_namespace_decl:
    s.namespace_decl (p.namespace_ (), p.prefix ());
    break;

    ...
  }
}
```

Namespace-Aware XML Filter Example

```
parser p (ifs,
          argv[1]
          parser::receive_default |
          parser::receive_namespace_decls);

for (...)
{
  switch (e)
  {
    ...

  case parser::start_namespace_decl:
    s.namespace_decl (p.namespace_ (), p.prefix ());
    break;

    ...
  }
}
```

High Level API

- Validation and error handling
- Attribute access
- Data extraction
- Content model processing
- Control flow support (member-aggregate, base-derived)

High Level API

```
<object id=”123”>
  <name>Lion’s Head</name>
  <type>mountain</type>

  <position lat=”-33.8569” lon=”18.5083”/>
  <position lat=”-33.8568” lon=”18.5083”/>
  <position lat=”-33.8568” lon=”18.5082”/>
</object>
```

Validation and Error Handling

```
parser p (ifs, argv[1]);

if (p.next () != parser::start_element ||
    p.qname () != "object")
{
  // error
}

...

if (p.next () != parser::end_element) // object
{
  // error
}
```

Validation and Error Handling

```
class parser
{
  void next_expect (event_type);
  void next_expect (event_type, const std::string& name);
};
```

Validation and Error Handling

```
parser p (ifs, argv[1]);

p.next_expect (parser::start_element, ”object”);

...

p.next_expect (parser::end_element); // object
```

Attribute Access

```
p.next_expect (parser::start_element, "object");

p.next_expect (parser::start_attribute, "id");
p.next_expect (parser::characters);
cout << "id: " << p.value () << endl;
p.next_expect (parser::end_attribute);

...

p.next_expect (parser::end_element); // object
```

Attribute Access

```
while (p.next () == parser::start_attribute)
{
  if (p.qname () == "id")
  {
    p.next_expect (parser::characters);
    cout << "id: " << p.value () << endl;
  }
  else if (...)
  {
  }
  else
  {
    // error: unknown attribute
  }

  p.next_expect (parser::end_attribute);
}
```

Attribute Access

```
class parser
{
  static const feature_type receive_elements;
  static const feature_type receive_characters;
  static const feature_type receive_attributes;

  ...
};
```

Attribute Access

```
class parser
{
  static const feature_type receive_elements;
  static const feature_type receive_characters;
  static const feature_type receive_attributes_map;
  static const feature_type receive_attributes_event;

  ...
};
```

Attribute Access

```
class parser
{
  const std::string&
  attribute (const std::string& name) const;

  std::string
  attribute (const std::string& name,
             const std::string& default_value) const;

  bool
  attribute_present (const std::string& name) const;
};
```

Attribute Access

```
class parser
{
  const std::string&
  attribute (const std::string& name) const;

  std::string
  attribute (const std::string& name,
             const std::string& default_value) const;

  bool
  attribute_present (const std::string& name) const;
};
```

Attribute Access

```
class parser
{
  const std::string&
  attribute (const std::string& name) const;

  std::string
  attribute (const std::string& name,
             const std::string& default_value) const;

  bool
  attribute_present (const std::string& name) const;
};
```

Attribute Access

```
p.next_expect (parser::start_element, ”object”);

cout << ”id: ” << p.attribute (”id”) << endl;

...

p.next_expect (parser::end_element); // object
```

Attribute Access

```
p.next_expect (parser::start_element, ”object”);

cout << ”id: ” << p.attribute (”id”) << endl;

...

p.next_expect (parser::end_element); // object
```

Data Extraction

```
class parser
{
  template <typename T>
  T value () const;

  template <typename T>
  T attribute (const std::string& name) const;

  template <typename T>
  T attribute (const std::string& name,
               const T& default_value) const;
};
```

Data Extraction

```
p.next_expect (parser::start_element, "object");

unsigned int id = p.attribute<unsigned int> ("id");

...

p.next_expect (parser::end_element); // object
```

Content Model

```
p.next_expect (parser::start_element, ”object”);
unsigned int id = p.attribute<unsigned int> (”id”);

p.next_expect (parser::start_element, ”name”);

...

p.next_expect (parser::end_element); // name

p.next_expect (parser::end_element); // object
```

Content Model

```
p.next_expect (parser::start_element, ”object”);
unsigned int id = p.attribute<unsigned int> (”id”);

p.next_expect (parser::start_element, ”name”);

...

p.next_expect (parser::end_element); // name

p.next_expect (parser::end_element); // object
```

```
<object id=”123”>
  <name>Lion’s Head</name>
```

Content Model

```
p.next_expect (parser::start_element, ”object”);
unsigned int id = p.attribute<unsigned int> (”id”);

p.next_expect (parser::start_element, ”name”);

...

p.next_expect (parser::end_element); // name

p.next_expect (parser::end_element); // object
```

```
<object id=”123”>
  <name>Lion’s Head</name>
```

Content Model

```
// p.next_expect (parser::start_element, ”name”);
cerr << p.next () << endl;
```

Content Model

```
// p.next_expect (parser::start_element, ”name”);
cerr << p.next () << endl;
```

```
<object id=”123”>#
##<name>Lion’s Head</name>
```

Content Model

```
namespace xml
{
  enum class content
  {           //  element   characters  whitespaces
    empty,    //    no          no        ignored
    simple,   //    no          yes       preserved
    complex,  //    yes         no        ignored
    mixed     //    yes         yes       preserved
  };
}
```

Content Model

```
namespace xml
{
  enum class content
  {           //  element   characters  whitespaces
    empty,    //     no          no         ignored
    simple,   //     no          yes        preserved
    complex,  //     yes         no         ignored
    mixed     //     yes         yes        preserved
  };
}
```

```
<empty name="A" id="1">
```

```
<empty/>
```

Content Model

```
namespace xml
{
  enum class content
  {           //  element   characters  whitespaces
    empty,    //    no          no         ignored
    simple,   //    no          yes        preserved
    complex,  //    yes         no         ignored
    mixed     //    yes         yes        preserved
  };
}
```

```
<simple name="B" id="2">
  sim
  ple
<simple/>
```

Content Model

```
namespace xml
{
  enum class content
  {           //  element   characters  whitespaces
    empty,    //    no          no        ignored
    simple,   //    no          yes       preserved
    complex,  //    yes         no        ignored
    mixed     //    yes         yes       preserved
  };
}
```

```
<complex name="C" id="3">
  <nested>...</nested>
  <nested>...</nested>
<complex/>
```

Content Model

```
namespace xml
{
  enum class content
  {           //  element   characters  whitespaces
    empty,    //    no          no         ignored
    simple,   //    no          yes        preserved
    complex,  //    yes         no         ignored
    mixed     //    yes         yes        preserved
  };
}
```

```
<mixed name="D" id="4">
  Mi<nested>...</nested>
  x<nested>...</nested>ed
<mixed/>
```

Content Model

```
namespace xml
{
  enum class content
  {           //  element   characters  whitespaces
    empty,    //    no          no        ignored
    simple,   //    no          yes       preserved
    complex,  //    yes         no        ignored
    mixed     //    yes         yes       preserved
  };

  class parser
  {
    void content (content);
  };
}
```

Content Model

```
p.next_expect (parser::start_element, "object");
p.content (content::complex);

unsigned int id = p.attribute<unsigned int> ("id");

p.next_expect (parser::start_element, "name"); // Ok.

...

p.next_expect (parser::end_element); // name

p.next_expect (parser::end_element); // object
```

Parsing Simple Content

```
p.next_expect (parser::start_element, ”name”);
p.content (content::simple);

p.next_expect (parser::characters);
string name = p.value ();

p.next_expect (parser::end_element); // name
```

Parsing Simple Content

```
std::string element ();

template <typename T>
T element ();

std::string element (const std::string& name);

template <typename T>
T element (const std::string& name);

std::string
element (const std::string& name,
         const std::string& default_value);

template <typename T>
T element (const std::string& name,
           const T& default_value);
```

Parsing Simple Content

```
std::string element ();

template <typename T>
T element ();

std::string element (const std::string& name);

template <typename T>
T element (const std::string& name);

std::string
element (const std::string& name,
         const std::string& default_value);

template <typename T>
T element (const std::string& name,
           const T& default_value);
```

Parsing Simple Content

```
std::string element ();

template <typename T>
T element ();

std::string element (const std::string& name);

template <typename T>
T element (const std::string& name);

std::string
element (const std::string& name,
         const std::string& default_value);

template <typename T>
T element (const std::string& name,
           const T& default_value);
```

Parsing Simple Content

```
p.next_expect (parser::start_element, "object");
p.content (content::complex);

unsigned int id = p.attribute<unsigned int> ("id");
string name = p.element ("name");

p.next_expect (parser::end_element); // object
```

Customizing Data Extraction

```
enum class object_type
{
  building,
  mountain,
  ...
};
```

Customizing Data Extraction

```
enum class object_type
{
  building,
  mountain,
  ...
};


object_type type = p.element<object_type> (”type”);
```

Customizing Data Extraction

```
namespace xml
{
  template <>
  struct value_traits<object_type>
  {
    static object_type
    parse (std::string, const parser&)
    {
      ...
    }

    static std::string
    serialize (object_type, const serializer&)
    {
      ...
    }
  };
}
```

Peeking

```
p.next_expect (parser::start_element, ”object”);
p.content (content::complex);
...

do
{
  p.next_expect (parser::start_element, ”position”);
  p.content (content::empty);

  float lat = p.attribute<float> (”lat”);
  float lon = p.attribute<float> (”lon”);

  p.next_expect (parser::end_element);

} while (p.peek () == parser::start_element);

p.next_expect (parser::end_element); // object
```

Peeking

```
p.next_expect (parser::start_element, ”object”);
p.content (content::complex);
...

do
{
  p.next_expect (parser::start_element, ”position”);
  p.content (content::empty);

  float lat = p.attribute<float> (”lat”);
  float lon = p.attribute<float> (”lon”);

  p.next_expect (parser::end_element);

} while (p.peek () == parser::start_element);

p.next_expect (parser::end_element); // object
```

```
parser p (ifs, argv[1]);
p.next_expect (
  parser::start_element, "object", content::complex);

unsigned int id = p.attribute<unsigned int> ("id");
string name = p.element ("name");
object_type type = p.element<object_type> ("type");

do
{
  p.next_expect (
    parser::start_element, "position", content::empty);

  float lat = p.attribute<float> ("lat");
  float lon = p.attribute<float> ("lon");

  p.next_expect (parser::end_element); // position
} while (p.peek () == parser::start_element);

p.next_expect (parser::end_element); // object
```

High Level Serializer API

```
class serializer
{
  void attribute (const std::string& name,
                  const std::string& value);

  void element (const std::string& value);

  void element (const std::string& name,
                const std::string& value);
};
```

High Level Serializer API

```
class serializer
{
  void attribute (const std::string& name,
                  const std::string& value);

  void element (const std::string& value);

  void element (const std::string& name,
                const std::string& value);
};
```

High Level Serializer API

```
class serializer
{
  void attribute (const std::string& name,
                  const std::string& value);

  void element (const std::string& value);

  void element (const std::string& name,
                const std::string& value);
};
```

High Level Serializer API

```
class serializer
{
  void attribute (const std::string& name,
                  const std::string& value);

  void element (const std::string& value);

  void element (const std::string& name,
                const std::string& value);
};
```

High Level Serializer API

```
class serializer
{
  template <typename T>
  void attribute (const std::string& name,
                  const T& value);

  template <typename T>
  void element (const T& value);

  template <typename T>
  void element (const std::string& name,
                const T& value);

  template <typename T>
  void characters (const T& value);
};
```

```
serializer s (cout, ”output”);
s.start_element (”object”);

s.attribute (”id”, 123);
s.element (”name”, ”Lion’s Head”);
s.element (”type”, object_type::mountain);

for (...)
{
  s.start_element (”position”);

  float lat (...), lon (...);

  s.attribute (”lat”, lat);
  s.attribute (”lon”, lon);

  s.end_element (); // position
}

s.end_element (); // object
```

Object Persistence

Object Persistence

```
<object name="Lion's Head" type="mountain" id="123">
  <position lat="-33.8569" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5082"/>
</object>
```

Object Model

```
enum class object_type {...};

class position
{
  ...

  float lat_;
  float lon_;
};

class object
{
  ...

  std::string name_;
  object_type type_;
  unsigned int id_;
  std::vector<position> positions_;
};
```

Object Model

```
class position
{
  position (xml::parser&);

  void serialize (xml::serializer&) const;
};

class object
{
  object (xml::parser&);

  void serialize (xml::serializer&) const;
};
```

Object Model

```
class position
{
  position (xml::parser&);

  void serialize (xml::serializer&) const;
};

class object
{
  object (xml::parser&);

  void serialize (xml::serializer&) const;
};
```

Object Parsing

```
position::
position (parser& p)
  : lat_ (p.attribute<float> ("lat")),
    lon_ (p.attribute<float> ("lon"))
{
  p.content (content::empty);
}
```

Object Parsing

```
object::
object (parser& p)
  : name_ (p.attribute ("name")),
    type_ (p.attribute<object_type> ("type")),
    id_ (p.attribute<unsigned int> ("id"))
{
  p.content (content::complex);

  do
  {
    p.next_expect (parser::start_element, "position");
    positions_.push_back (position (p));
    p.next_expect (parser::end_element);

  } while (p.peek () == parser::start_element &&
           p.name () == "position");
}
```

Object Parsing

```
object::
object (parser& p)
  : name_ (p.attribute ("name")),
    type_ (p.attribute<object_type> ("type")),
    id_ (p.attribute<unsigned int> ("id"))
{
  p.content (content::complex);

  do
  {
    p.next_expect (parser::start_element, "position");
    positions_.push_back (position (p));
    p.next_expect (parser::end_element);

  } while (p.peek () == parser::start_element &&
           p.name () == "position");
}
```

Optional Attributes

```
object::
object (parser& p)
  : ...
    type_ (p.attribute (”type”, object_type::other))
    ...
```

Object Serialization

```
void position::serialize (serializer& s) const
{
  s.attribute (”lat”, lat_);
  s.attribute (”lon”, lon_);
}

void object::serialize (serializer& s) const
{
  s.attribute (”name”, name_);
  s.attribute (”type”, type_);
  s.attribute (”id”, id_);

  for (const auto& p: positions_)
  {
    s.start_element (”position”);
    p.serialize (s);
    s.end_element ();
  }
}
```

Object Persistence

```
parser p (ifs, argv[1]);
p.next_expect (parser::start_element, ”object”);
object o (p);
p.next_expect (parser::end_element);


serializer s (cout, ”output”);
s.start_element (”object”);
o.serialize (s);
s.end_element ();
```

Object Persistence

```
parser p (ifs, argv[1]);
object o (p);


serializer s (cout, ”output”);
o.serialize (s);
```

Object Model Root

```
object::
object (parser& p)
{
  p.next_expect (
    parser::start_element, "object", content::complex);

  name_ = p.attribute ("name");
  type_ = p.attribute<object_type> ("type");
  id_ = p.attribute<unsigned int> ("id");

  ...

  p.next_expect (parser::end_element);
}
```

Object Model Root

```
void object::
serialize (serializer& s) const
{
  s.start_element (”object”);

  ...

  s.end_element ();
}
```

Object Model Root

```
object::
object (parser& p)
{
  p.next_expect (
    parser::start_element, "object", content::complex);

  name_ = p.attribute ("name");
  type_ = p.attribute<object_type> ("type");
  id_ = p.attribute<unsigned int> ("id");

  ...

  p.next_expect (parser::end_element);
}
```

Inheritance

Inheritance

```
<elevated-object name="Lion's Head" type="mountain"
                 units="m" id="123">
  <position lat="-33.8569" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5083"/>
  <position lat="-33.8568" lon="18.5082"/>

  <elevation val="668.9"/>
  <elevation val="669"/>
  <elevation val="669.1"/>
</elevated-object>
```

Inheritance

```
enum class units {...};

class elevation {...};

class elevated_object: public object
{
  ...

  units units_;
  std::vector<elevation> elevations_;
};
```

Inheritance Parsing

```
elevated_object::
elevated_object (parser& p)
  : object (p),
    units_ (p.attribute<units> ("units"))
{
  do
  {
    p.next_expect (parser::start_element, "elevation");
    elevations_.push_back (elevation (p));
    p.next_expect (parser::end_element);

  } while (p.peek () == parser::start_element &&
           p.name () == "elevation");
}
```

Inheritance Parsing

```
elevated_object::
elevated_object (parser& p)
  : object (p),
    units_ (p.attribute<units> ("units"))
{
  do
  {
    p.next_expect (parser::start_element, "elevation");
    elevations_.push_back (elevation (p));
    p.next_expect (parser::end_element);

  } while (p.peek () == parser::start_element &&
           p.name () == "elevation");
}
```

Inheritance Parsing

```
elevated_object::
elevated_object (parser& p)
  : object (p),
    units_ (p.attribute<units> ("units"))
{
  do
  {
    p.next_expect (parser::start_element, "elevation");
    elevations_.push_back (elevation (p));
    p.next_expect (parser::end_element);

  } while (p.peek () == parser::start_element &&
           p.name () == "elevation");
}
```

Inheritance Serialization

```
void elevated_object::
serialize (serializer& s) const
{
  object::serialize (s);

  s.attribute (”units”, units_);

  for (const auto& e: elevations_)
  {
    s.start_element (”elevation”);
    e.serialize (s);
    s.end_element ();
  }
}
```

Inheritance Serialization

```
void object::
serialize_attributes (serializer& s) const
{
  s.attribute (”name”, name_);
  s.attribute (”type”, type_);
  s.attribute (”id”, id_);
}

void object::
serialize_content (serializer& s) const
{
  for (const auto& p: positions_)
  {
    s.start_element (”position”);
    p.serialize (s);
    s.end_element ();
  }
}
```

Inheritance Serialization

```
void object::
serialize (serializer& s) const
{
  serialize_attributes (s);
  serialize_content (s);
}
```

Inheritance Serialization

```
void elevated_object::
serialize_attributes (serializer& s) const
{
  object::serialize_attributes (s);
  s.attribute (”units”, units_);
}

void elevated_object::
serialize_content (serializer& s) const
{
  object::serialize_content (s);

  for (const auto& e: elevations_)
  {
    s.start_element (”elevation”);
    e.serialize (s);
    s.end_element ();
  }
}
```

Inheritance Serialization

```
void elevated_object::
serialize (serializer& s) const
{
  serialize_attributes (s);
  serialize_content (s);
}
```

API Summary

- `next_expect()`
- Detection of missing/extra attributes
- Validation of content models
- One-call attributes and simple content elements
- Attribute map with extended lifetime
- Whitespace processing
- Data extraction

API Summary

- `next_expect()`
- Detection of missing/extra attributes
- Validation of content models
- One-call attributes and simple content elements
- Attribute map with extended lifetime
- Whitespace processing
- Data extraction

What do you think?

Implementation Details

`libstudxml`

- Open-source (MIT)
- Cross-platform (autotools, VC9-12 projects)
- Small and dependency-free

Implementation Details

Conforming XML 1.0 Parser?

Implementation Details

Conforming XML 1.0 Parser?

Based on tried and tested Expat

Implementation Details — Parser

- Includes Expat source code as implementation detail

Implementation Details — Parser

- Includes Expat source code as implementation detail
- Push-to-Pull conversion via parser suspension

Implementation Details — Parser

- Includes Expat source code as implementation detail
- Push-to-Pull conversion via parser suspension
- 35% Push-to-Pull performance penalty

Implementation Details — Parser

- Includes Expat source code as implementation detail
- Push-to-Pull conversion via parser suspension
- 35% Push-to-Pull performance penalty
- 37 MBytes/s throughput on this oldish laptop

Implementation Details — Serializer

- Uses (heavily customized) Genx library
- Genx is a small C XML Serializer by Tim Bray

Implementation Details

- Based on mature XML parser and serializer
- Has been used in production in ODB

What's Next?

- http://codesynthesis.com/projects/libstudxml/
- Unix: `./configure && make`
- Windows: projects/solutions for VC++ 9, 10, 11, 12
- API Documentation
- Examples (`performance`, `hybrid`)